Introduction

This assignment is designed to give you experience with reuse in Java, by employing inheritance to extend and modify the behaviour of Java classes. You will use the *SimpleURLReader* class from our cs1 package. To complete the assignment, you do not need to know how the *SimpleURLReader* class works, you just treat it like any other Java class. One of the key points of this lab is to demonstrate that you can add to and modify the behaviour of an existing class, even without having its source code. Now that is neat!

Setup

Create your lab02 project from the CS101 console template, as usual, then download this cs1 package (right click and save it) to any convenient location outside of your lab02 project folder (e.g. the private\cs102 folder that contains all your labXX folders.) You now need to add the package to your project's *classpath* so Java can find and use it.

To do this **in DrJava:** Choose Project | ProjectProperties then, in the "Extra Classpath" option, select "Add" and navigate to the cs1.jar file you just downloaded. Select it and click OK everywhere to complete the process. All being well you should now be able to *import cs1.SimpleURLReader;* into your classes.

> ***Aside:*** You might be tempted to do this assignment without making use of (DrJava) projects ~by setting the classpath in your computer's global environment settings or from the Edit | Properties, ExtraClassPath option in DrJava~ but do not do so. Instead, **use projects to organise your work**. This is a key software engineering skill that will make your life much easier in the future, so make sure you learn how to do it now!

Exercises

Do the following in order:

(a) Write a test program that will read the contents of this url and print its contents and the number of lines it contains, on the console. To read the contents of a url you can use the *SimpleURLReader* class (see above!) Its constructor takes the desired *url* as a *String*, for example "http://www.cs.bilkent.edu.tr/~david/housman.txt". The class has only two methods: *getPageContents()* that returns the contents of the url (the webpage) as a *String*, and *getLineCount()* that returns an *int* corresponding to the number of lines read from the url.

(b) Design, implement and test a new class, *MySimpleURLReader*, that extends the *SimpleURLReader* class, adding two methods to it: *getURL()* that returns the url String used to create the SimpleURLReader object, and *getName()* that returns the filename part of the url, that is, the part of the url following the last '/' character ("housman.txt" in the previous example). Once this is working, fix the bug in SimpleURLReader's *getPageContents()* method whereby the String "null" is added to the beginning of the String it returns. Do this by overriding the corresponding method in your new sub-class.

(c) A customer wants to be able to print the contents of this other url (an html version of the original plain text) and have it appear as in the part (a) without any of the html code in it! Clearly, it is necessary to read the contents of the url and then filter out the html code so that only the visible text is left. Rather than write an entirely new class from scratch, you realise that the *MySimpleURLReader* class does most of what you want and so decide to use it. Design, implement & test a new class, *HTMLFilteredReader*, that extends MySimpleURLReader. Its *getPageContents()* method should return only the text, without the html. A new method, *getUnfilteredPageContents()* can be called to return the original page complete with html codes. Assume that anything between "<" and ">" is html code and should be omitted from the filtered output. Try to solve this problem using just charAt(i) and String concatenation.

(d) The customer is impressed and immediately asks you to add a method that computes the overhead due to the html code (the percentage increase in size between the html and no-html contents.) They also want a list of the url's that the page links to! A little research shows that html links have the form ***href="link_url".*** This time, try using the String class methods indexOf & substring, to extract all of the ***link_url***'s and put them into an ArrayList. Return this as the result of a *getLinks()* method. Add these methods by sub-classing *HTMLFilteredReader* in a class called *XHTMLFilteredReader*. Modify your test program to exercise these new facilities. Experiment with Java's extended type checking mechanism by calling the methods using variables of *MySimpleURLReader*, *HTMLFilteredReader* and *XHTMLFilteredReader* types. Note: the previous url's do not have any html links in them, so try using this url for testing this new class.

(e) Design and implement a simple menu-driven program that will maintain a collection of *MySimpleURLReader* objects. The main menu should have three options: (1) Enter the url of the poem to add to collection, (2) List all poems in the collection, and (3) Quit. Option 2 should display only the index number & (file) name for each of the poems. The user should then be able to enter the index number of a poem to view it (and then return to the same list.) If the user enters the last index number + 1 they should be returned to the main menu, anything else should be ignored.

In option 1, if the user enters the url of a text file you should create a *MySimpleURLReader* object and add it to the collection, whereas if they enter the url of an html file you should create an *HTMLFilteredReader* object and add it to the collection. You should always call the *getPageContents()* method of the corresponding object to view it from option 2. If you have done everything properly, you should always see the non-html version of the poem... that's neat, that's polymorphism!

(f) *...coming soon!*
